RV-ECU: Maximum Assurance In-Vehicle Safety Monitoring

Philip Daian, Bhargava Manja
Runtime Verification Inc., USA

Shin'ichi Shiraishi
Toyota Info Technology Center Inc., USA

Akihito Iwai
DENSO International America Inc., USA

Grigore Rosu
University of Illinois at Urbana-Champaign and Runtime Verification Inc., USA

ABSTRACT

The Runtime Verification ECU (RV-ECU) is a new development platform for checking and enforcing the safety of automotive bus communications and software systems. RV-ECU uses runtime verification, a formal analysis subfield geared at validating and verifying systems as they run, to ensure that all manufacturer and third-party safety specifications are complied with during the operation of the vehicle. By compiling formal safety properties into code using a certifying compiler, the RV-ECU executes only provably correct code that checks for safety violations as the system runs. RV-ECU can also recover from violations of these properties, either by itself in simple cases or together with safe message-sending libraries implementable on third-party control units on the bus. RV-ECU can be updated with new specifications after a vehicle is released, enhancing the safety of vehicles that have already been sold and deployed.

Currently a prototype, RV-ECU is meant to eventually be deployed as global and local ECU safety monitors, ultimately responsible for the safety of the entire vehicle system. We describe its overall architecture and implementation, and demonstrate monitoring of safety specifications on the CAN bus. We use past automotive recalls as case studies to demonstrate the potential of updating the RV-ECU as a cost effective and practical alternative to software recalls, while requiring the development of rigorous, formal safety specifications easily sharable across manufacturers, OEMs, regulatory agencies and even car owners.

INTRODUCTION

Modern automobiles are highly computerized, with 70 to 100 complex and interconnected electronic control units responsible for the operation of automotive systems, and roughly 35 to 40 percent of the development cost of modern automobiles going towards software. In the next 10 years this number is expected to jump to between 50 and 80 percent, and even higher for hybrid vehicles. This will only be more true with the advent of autonomous vehicles [1, 2].

It is not surprising, then, that the automotive industry suffers from nearly every possible software fault and resulting error. Many related stories have recently been featured on the news, including cases where cars are hacked and remotely controlled, including brakes and the engine, completely ignoring driver input. In some cases prior physical access to the car was needed, in others the car was not even touched. Massive automobile recalls in the past few years have been due to software bugs, costing billions [3, 4, 5, 6, 7, 8, 9]. Moreover, almost 80 percent of car innovations currently come from computer software, which has therefore become the major contributor of value in cars [1]. As software becomes more and more integral to the function and economics of vehicles, the safety and security of car software has taken center stage.

LIMITATIONS OF CURRENT APPROACHES Traditional software development quality processes rely on static analysis tools and techniques to improve the quality, security and reliability of their code. Static

analysis tools analyze software code against a set of known rules and heuristics and notify the operator of warnings and violations. Nearly all companies developing a reasonably large code base use code quality tools. The reader interested in how static analysis tools perform on automotive-related software is referred to [10]. Even with all of the resources spent on these tools, software is still full of bugs and reliability weaknesses. This may be fine when the software is running on something as simple as a cell phone, or a laptop computer that your child uses for homework, but this is unacceptable at best, and dangerous at worst, when the software runs in an automobile.

Model checking [11] is a complementary approach that has found some use in the automotive industry. While rigorous and thorough, this approach suffers from serious drawbacks that make its use impractical. Besides the infamous "state explosion" problem, the most significant drawback of model checking is the issue of model faithfulness. Models being used must be correct with regards to the system being inspected and the environment it operates in. With the complexity of modern software and hardware systems, and the (often) specific nature of the models involved, great care must be taking in validating the model itself as well as the system with regards to the model. This is an extremely error prone and time intensive process. A previous comparison of model checking to static analysis by a team investigating model checking tools found that issues in the model itself caused model checking to miss five errors caught by static analysis, concluding that "the main source of false negatives is not incomplete models, but the need to create a model at all. This cost must be paid for each new checked system and, given finite resources, it can preclude checking new code." [12]. Besides the model, the tool itself must also be trusted to properly verify the properties over the model, requiring either a highly-audited open source tool or another source of high confidence in the tool itself.

The portability of these models and specifications is also dubious: any changes in the underlying system require a correct change in the model, a non-trivial process that must be repeated often for complex systems [11]. Equivalent specifications can thus have different meanings based on the models being used.

While this does not matter if the model is specific to some standard, such as a programming language [13], with many tools and applications of model checking this is not the case [14, 15]. So, while expressive, models can be complex and non-portable. Overall, while model checking has the potential for detecting deep and subtle errors, the requirement for a model introduces many restrictions and complexities that make the tools difficult to manage and integrate effectively into most engineering teams, restricting their use to teams with high levels of formal expertise and critical applications requiring the maximum possible assurance, thus preventing widespread adoption by the automotive industry as a whole.

ENABLING SAFETY STANDARDIZATION Another hurdle on the path to greater automotive safety is the lack of standardized automotive safety specifications. Because many specifications are informally expressed and never formalized, communication between Tier 1 suppliers and their OEM partners is often incomplete with regards to safety, producing components that may behave unpredictably in the system as a whole. Moreover, formalizations that exist tend to be difficult or impossible to port between Tier 1 suppliers. One clear industry need stemming from verification-based development methodologies is the need for portable formal safety specifications. Specifications should be expressed in lightweight formalisms that are easy to understand and communicate, and should stay separate from the particular verification approach that is employed for their checking.

Lastly, we observe that in currently developed automotive systems, both the safety and the functionality of the system and its components are considered and implemented together, as part of the same development process. Because safety and functionality are necessarily related to each other, this appears to be logical. However, this intermixing of safety checks in components that are primarily functional represents a violation of the maximum possible separation of concerns in an ideal system architecture, in which safety would be considered and implemented separately from the desired functionality, allowing for a clean separation that promotes both safety testing and rigorous reasoning about safety properties.

As an alternative to static verification methods, runtime verification makes possible easy standardization of rigorous formal safety specifications and clean separation between functionality and safety components of systems.

RUNTIME VERIFICATION

Runtime verification is a system analysis and approach that extracts information from the running system and uses it to assess satisfaction or violation of specified properties and constraints [16]. Properties are expressed formally, as finite state machines, regular expressions, linear temporal logic formulas, etc. These formal requirements are used to synthesize monitors, and the existing code base is automatically

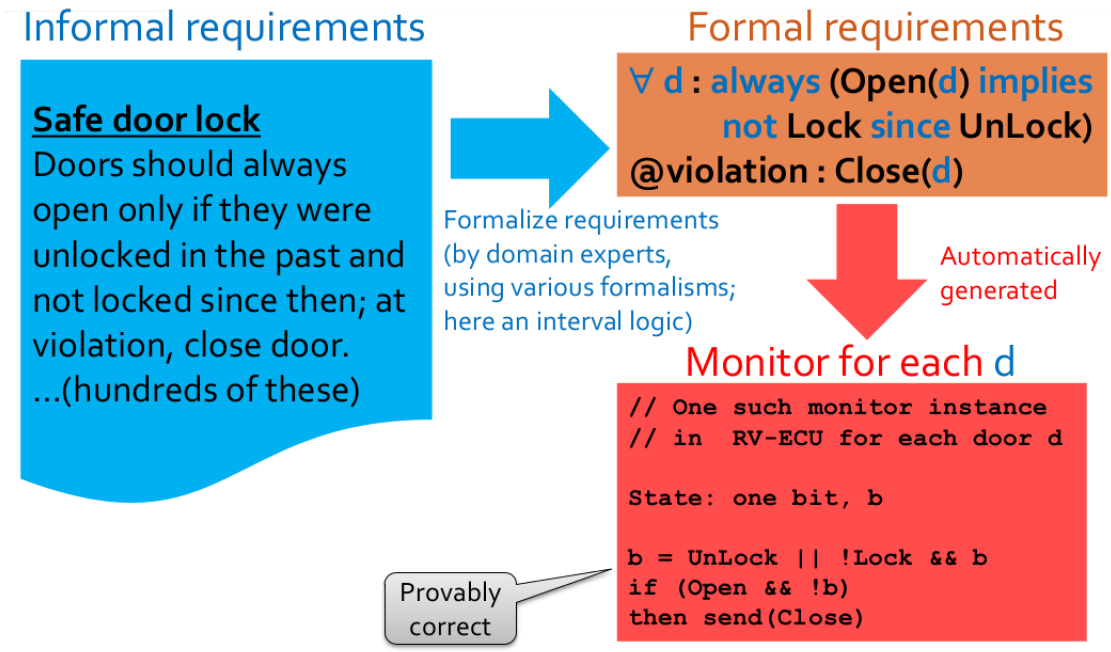

Figure 1: Example automotive specification being compiled into code that enforces it at runtime through RV-ECU

instrumented with these monitors. Runtime verification can be used for many purposes, including policy monitoring, debugging, testing, verification, validation, profiling, behavior modification (e.g., recovery, logging), among others. In the development cycle, runtime verification can be used as a bug finding tool, a testing approach, a development methodology focusing on the creation of formally rigorous specifications, while in a production system it can be used as a component responsible for enforcing a set of safety requirements on a system to preserve its global safety during operation.

Ideally, developers and software quality managers would like to validate a system prior to its operation and release, in the coding and testing phases of the software engineering lifecycle. This would allow maximum assurance in the performance of the deployed system before release, increasing software dependability. However, as previously discussed, static validation methods such as model-checking [17] suffer from limits preventing their use in real large-scale applications. For instance, those techniques are often bound to the design stage of a system and hence they are not shaped to face-off specification evolution. Even when static analysis techniques do scale, they are limited by the properties they can check, and may not be able to check interesting behavioral properties. Thus, the verification of some properties, and elimination of some faults, have to be complemented using methods relying on analysis of system executions.

Figure 1 shows an example of an automotive safety specification being compiled to code that enforces it at runtime using the technology underlying RV-ECU. The specification is called “safe door lock”, and is first stated in plain English informally, as safety requirements are currently expressed. This is translated to a formal requirement manually by domain experts, as shown in the orange box using linear temporal logic: it is always the case that a valid door open event implies that there has not been a door lock since the last unlock; a recovery action is attached that closes the door when a violation is detected (violation handler).

Previous efforts in the runtime verification field have focused on the development of formalisms appropriate for specifying expressive properties while synthesizing efficient monitors [18, 19, 20, 21, 22, 23], steering program and system executions to obtain desirable behaviors [24], combining runtime verification with other approaches including static analysis [25], minimizing runtime overhead to make monitoring deployed systems practical [22, 26], and integrating runtime verification with existing projects automatically through both binary and source instrumentation, often leveraging aspect-oriented programming [27, 28].

Because runtime verification is a relatively new field, the number of practical and commercial applications of the technology is less substantial than that of static analysis tools, model checkers, or deductive program verifiers. There have, however, been some practical applications of the theory of runtime enforcement for program safety and security [29, 30, 31], or to enforce access control policies at system runtime [32,

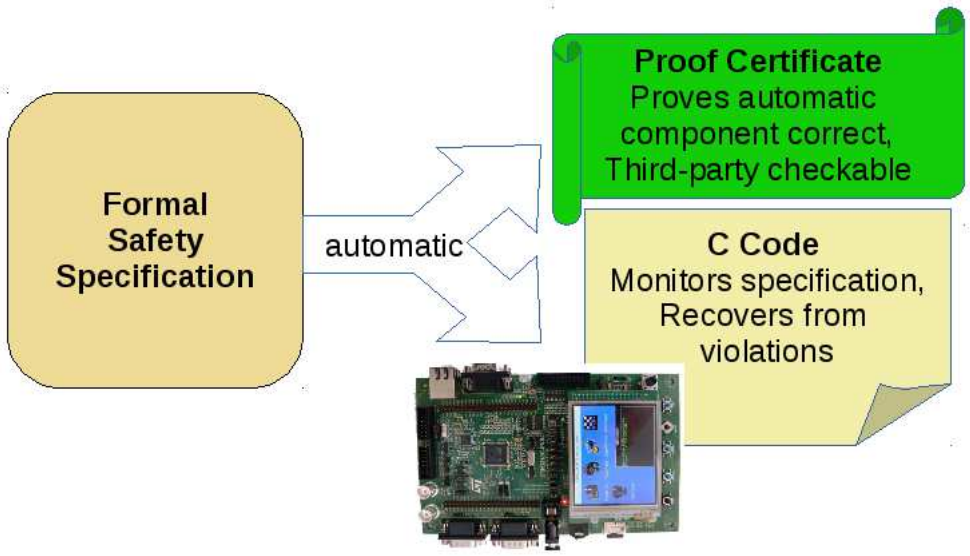

Figure 2: RV-ECU system, applying automatic certifying compilation of safety specifications

33]. Runtime verification has also been applied to mobile applications to provide fine-grained permissions controls and enforce device security policies [34].

RV-ECU: A VEHICLE SAFETY ARCHITECTURE

Because the automotive industry develops some of the most widely deployed safety critical software of any industry, it represents an ideal context where the benefits of runtime verification can make a significant difference.

Towards this goal we introduce RV-ECU, a development platform, also referred to as a “workbench” or a “system” in the paper, for checking and enforcing the safety of automotive bus communications. For brevity, whenever the context is non-ambiguous we take the freedom to use the same name “RV-ECU” for any of its components or even for other components that make use of code produced using RV-ECU.

At its core, RV-ECU consists of a compiler from formally defined safety specifications to monitoring code running on embedded control units. The safety specifications can be designed in any known mathematical formalism, with RV-ECU providing a plugin-based system to enable the development of custom formalisms for the specific automotive needs. Currently, some supported specifications languages include finite state machines, regular expressions, linear temporal logic, and context-free grammars.

To provide a clearer picture of what RV-ECU is and what it can do, we will explain its use, from end to end and step by step:

1. Trained personnel use the formalism of their choice to specify a safety property. Take, for example, the property that the windshields being on the fastest setting implies the headlights are on the brightest setting. The specification includes ”recovery actions” to take if the property is violated, to return to a safe state
2. The RV-ECU compiler is invoked on the formal definition, creating monitors and proof objects that certify the monitors are correct with regards to the specification
3. The original code base is instrumented, either automatically or manually, with calls to monitors are relevant call sites. Thus, the safety checking/recovery functionality is cleanly orthogonal to other functionality considerations.

Figure 2 shows an overview of the RV-ECU methodology, which takes formal specifications as input and from them automatically outputs code checking these specifications as well as a proof object certifying the correctness of this code over the mathematical semantics of the specification formalism and of the underlying programming language. Thus, the code output by the RV-ECU compiler provides correctness proof certificates of the monitoring code as well as of the recovering code which is executed when the original specification is violated. These certificates can be checked in third party theorem proving software, providing the maximum known assurance guarantees that the code running on-device implements the given safety specifications and their recovery handlers.

The benefits of the RV-ECU approach are numer-

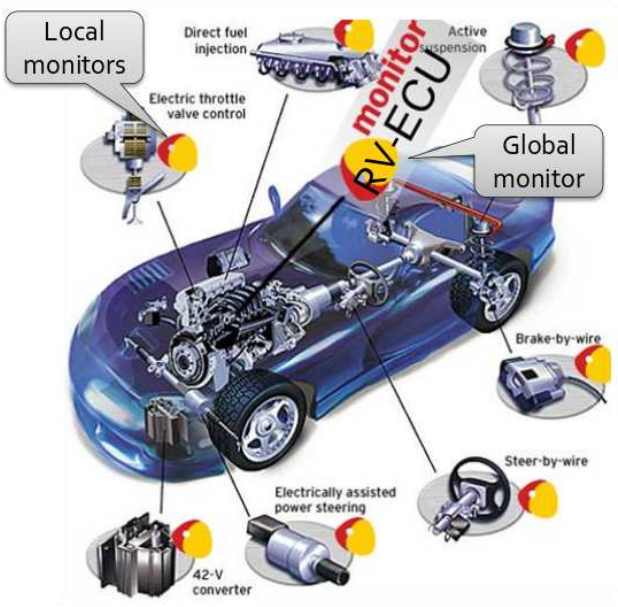

Figure 3: RV-ECU running both globally and locally, checking and enforcing vehicle system-wide safety.

ous. RV-ECU's compatibility with many formalisms and its function as a compiler to monitors completely decouples considerations of functionality from those of safety. Automotive software engineers are free to focus their efforts on code that enhances the functionality of software systems aboard the automobile, while safety engineers can focus on formalizing and testing safety properties. This decoupling allows for the development of modular and reusable safety formalisms that can easily be shared between automotive suppliers and OEMs. This can be revolutionary, as it ensures compatibility in safety specifications between OEMs and Tier 1 suppliers. It even makes possible a standardized database of formal safety properties maintained and updated by state regulatory bodies.

GLOBAL AND LOCAL MONITORING Figure 3 shows the RV-ECU system running on a vehicle. It is important to note that RV-ECU can be applied in two places: the generated monitoring code can either run on a separate control unit to monitor the global traffic on the bus, or be integrated within an existing control unit (e.g., the power steering ECU) to prevent it from taking unsafe actions. We therefore distinguish two categories of monitors, with "global" monitors observing the bus on a dedicated ECU and "local" monitors observing the bus from the perspective of an existing ECU responsible for some functionality.

These global and local monitors can then further communicate to increase their effectiveness. When used together, the global monitors can track the state of the entire vehicle system, with local monitors tracking only the state important to a particular controller. By communicating over the CAN bus, they are able to share messages and commands, and the global monitor is able to instruct the local monitors to block or modify messages they may otherwise allow.

For simple testing and safety specifications involving one component, local monitoring can be used. With complex or resource-intensive properties involving multiple components, global monitoring can be used. A combination of these approaches can be applied both in the testing cycle and the production automobile, spanning the extremes between global monitoring of the entire system only with untrusted code running on individual components and local monitoring of specific components only with no global specifications or dedicated hardware. This flexibility allows OEMs and Tier 1 suppliers to choose how and where they apply the runtime verification technology, allowing for incremental rollouts of local monitors at first followed by the eventual implementation of a global monitor, or vice versa.

Figure 4 shows the ideal RV-ECU deployment, with all ECUs on the bus containing local monitors and a global monitor attached to the full system. In this example, no communication can flow between untrusted, manually-written code implementing functionality (highlighted in yellow) and the vehicle bus without approval from high-assurance, provably correct, automatically generated code implementing the safety specifications of the vehicle.

The use of RV-ECU therefore protects the overall safety of the system from both malfunctioning controllers and malicious accesses (hackers), maintaining a set of safety invariants specified rigorously during the development of the vehicle. Moreover, the safety monitoring code generated by RV-ECU uses state-of-the-art techniques and algorithms developed by the runtime verification community specifically aimed at minimizing runtime overhead.

CERTIFIABLE CORRECTNESS

As previously mentioned, the code generated by the RV-ECU system from safety specifications additionally carries proof certificates. Proof certificates are mathematical objects expressed as objects in the Coq automated theorem proving assistant [35], a proof assistant that has been widely successfully applied to detect security flaws in popular software [36], prove mathematical theorems [37], and create and prove the most complete currently certified C compiler [38].

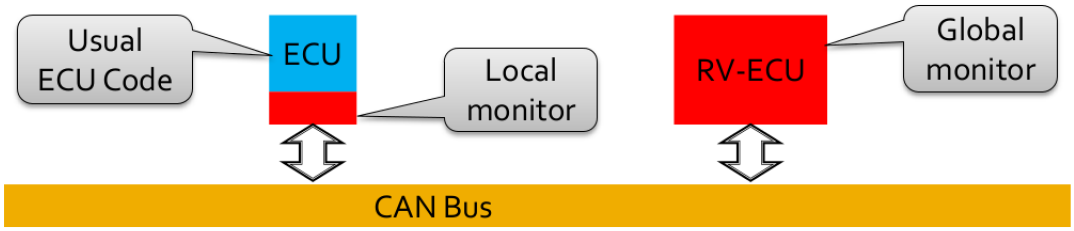

Figure 4: RV-ECU protecting the CAN bus from unsafe messages

```
void step(int *state, int event) {
    if(*state == 0) {
        if(event == 0) {
            *state = 1;
        } else if(event == 1) {
            *state = 2;
        }
    } else if(*state == 1) {
        if(event == 0) {
            *state = 3;
        } else if(event == 1) {
            *state = 0;
        }
    }
}
```

```
require "../../kernelc.k"

module MINIMUM-SPEC
  imports KERNELC

rule
  <fun>... FUN:Map ...</fun>
  <k>
    main(.Expressions)
  =>
    tv(void, undef)
  </k>

endmodule
```

Figure 5: FSM Transition for Monitor and Formal Specification Verying Program

The proof objects we provide mathematically prove that the code we generate correctly implements the specification inputs provided, with regards to the mathematical formal semantics of the programming language itself. These proofs are machine checkable by third party theorem proving tools including but not limited to Coq, providing multiple independent sources of assurance that the generated code is rigorously correct. Such proof objects can also be used in the context of certifying vehicle safety, with their formal rigor providing the maximum known standards for software development in the context of rating development assurance in standards like ISO 26262.

To demonstrate such proof certificates, we have proved the simple finite state monitor shown in Figure 5 using our in-house verification technology, the K Framework. The figure shows the transition function of the finite state monitor, in this case a simplified transition function implemented in KernelC. It also shows a proof specification for the program, showing that we would like to prove that the main function rewrites to a void value. This void value in our semantics of KernelC indicates that the program execution terminated with no error.

In this example we prove that the program, when instrumented with the monitoring code, will never reach a state we define as an error state. Our proof thus requires both the monitoring code and the program being instrumented to complete, and proves the correctness of our recovery action with regards to our property as well.

We reached several challenges in defining the target for such a proof of correctness. Our initial operating assumption was that a monitor is correct when, for all possible event traces, the monitoring code will end in an internal state consistent with the property it claims to monitor. This definition does not however take into account the steering aspect of runtime monitors: because runtime monitors are interactive in the program, they have the ability to modify the program's path as they run and execute recovery actions. Some event traces that may be captured in the property may then be unreachable by the monitor, which actively steers the program away from such traces through recovery actions.

Another key point that is not captured by the above definition of monitor correctness is instrumentation. If it is possible that a monitor may miss events, any guarantees provided by the above definition are entirely meaningless. If a monitor process events that

did not occur, the same is true. Our final correctness notion somehow thus should include the notion of instrumentation.

Naturally, we need to draw a line somewhere to create a trusted assurance base usable in what we seek to prove. We believe the compiler boundary is the ideal place for that separation in our work: we assume that the compiler will not introduce any behavior into the system inconsistent with the semantics of the programming language it takes as input, and we assume a lack of hardware faults. Increasing the assurance of both areas is a separate research task with ongoing academic research being pursued in certified compilation [39] [40] and trusted hardware systems [41] [42] [43].

Because of the need to include instrumentation in our correctness guarantees, we must therefore naturally consider the program being executed (which is the instrumentation site for the monitor). We believe the correct definition of a correct (Monitor, Program) pair is then that the Program, when instrumented with the Monitor, will never reach any state violating the specified safety property. Thus, we must prove that the safety property itself holds over the program, which is equivalent to proving that the safety property holds in the unmonitored program. This is exactly the proof we created through our K Framework verification technology, using the inputs shown in Figure 5 as well as the C code of the same program being executed.

We are continuing to explore alternate correctness definitions and guarantees, and are developing the infrastructure to provide Coq-verifiable certificates for all potential guarantees.

RV-ECU COMPARED: OTHER RV EFFORTS

There is a fair amount of supporting work in applications of runtime verification to critical systems, embedded systems, and automotive systems. The most similar to our work is by Kane, wherein runtime CAN bus monitors for a range of properties are implemented [44]. However, his work is not viable for industrial use. His chosen development board, an STFF4-Discovery microcontroller, does not include a CAN transceiver, so he added a breadboard for CAN read/write functionality. By contrast, RV-ECU is an integrated hardware/software system, with the hardware ECU is capable of being used in a vehicle without modification, as we have shown in our demo.

Other work in runtime monitoring for ultra critical systems and hard real time monitoring also falls short in its applicability to the automotive sector. One representative example is the Copilot system from Galois, Inc [45] [46]. Though it, too, is a compiler from formalisms to embedded C monitors, it requires a completely custom formalism to specify properties, and moreover requires expertise in Haskell, a niche language, and the use of the custom Copilot embedded domain specific language. In contrast, RV-ECU's modular plugin system allows specification in arbitrary formalisms and knowledge only of C, the language of choice in automotive software.

Many other runtime systems for automotive monitoring require specialized hardware or hardware modifications to ECUs [47] [48]. In contrast, RV-ECU can function completely in software, as the actual monitoring ECU is a completely optional addition to the CAN network. Of all systems we have seen, RV-ECU is the most flexible, adaptable, and general system for runtime monitoring of automobiles. It also currently has the most rigorous infrastructure for proving monitors correct, giving some additional assurance that the code on-device behaves as intended. We intend to continue extending these features as they become relevant to our partners and customers.

RECALLS AND RV-ECU, A CASE STUDY

One of the key problems in the automotive industry we believe will be helped by the RV-ECU technology is a reduction in the required number of software recalls, as well as a quicker and less costly response when recalls must be performed. To demonstrate this application of our technology, we consider previous software-caused recalls in the automotive industry.

We do not have to look far to find good examples. Just a few months ago two security researchers unveiled an exploit that gave them full, remote access to the CAN bus of the Chrysler Jeep Cherokee [49]. The two researchers found an unauthenticated open port on the car's Uconnect cellular network interface, and used this foothold, as well as the fact that firmware binaries were unsigned, to update the car's networking hardware over the air with a backdoored firmware that gave them the ability to sniff CAN messages.

An unauthenticated SPI line between their backdoored chip and a CAN controller allowed them to write arbitrary messages over the CAN bus. Their control over the car was near total - they demonstrated complete wireless control over braking, the sound system, the driver display, door locks, AC, windshield wipers, steering (in reverse), and transmission [50]. They publicized their research, after disclosing the issue to regulators the car companies involved, with a dramatic article in Wired magazine.

A Wired journalist took a spin in a hacked car, which the researchers remotely drove into a ditch [49]. This announcement created waves both in the automotive industry and among the general public, and continues to inspire both continued media and public discussion, as well as safety legislation. The hack led to a recall of 1.4 million cars, the proposal of new vehicle cyber safety regulation in Congress, and a $400 million drop in Fiat Chrysler's market cap [51].

This incident highlights the deficiencies of the automotive industry with regards to safety, and the adverse effects of informal software engineering methodology on both end consumers and the bottom line. Fiat was lucky in that the two security researchers chose to disclose this exploit. More exploits along the same vein are sure to exist. How then can runtime verification technology help automobile manufacturers improve vehicular safety?

In this specific case, RV-ECU could have come into play in multiple ways. The researchers mention in their Blackhat conference paper that, to their surprise, while the Jeep's firmware update mechanism was designed to be operated via the dashboard display, nothing prevented them from sending firmware update commands over the air, without authentication. This entire attack approach would have been rendered invalid with one simple global safety property formalizing the requirement that firmware updates must be driven from the dashboard display only.

This does not, however, deal with the more fundamental problem that CAN traffic is unauthenticated and multicast. This means that all an attacker needs to do to gain control over an automobile is gain access to the CAN bus and impersonate legitimate ECUs. Through local and global monitors, RV-ECU easily allows the implementation of authentication and authorization protocols as lightweight formalisms completely orthogonal to the functionality of the software components. In other words, the engineers developing the code that achieves the desired functionality of the ECU need not worry about authentication, that being added automatically by RV-ECU. This achieves a separation of concerns that makes authentication and authorization simpler and more portable.

Even if the researchers found a way past that, proper formalization of vehicle safety would prevent many of their attacks from taking place, even if they could impersonate legitimate ECUs. We cannot assume that the automotive industry will be able to correct all security vulnerabilities that could lead through compromise through traditional testing and analysis: even in the payments industry, where security has been a key focus and source of spending and concern, recent studies have concluded that the complexity of modern software systems makes breaches virtually impossible to avoid [52]. Such a conclusion likely also applies to automotive, with increasingly connected and complex systems implying that the elimination of all security-sensitive software errors and user error is unlikely if not impossible. We must thus manage the risks entailed by a compromise, providing a trusted hardware base that is minimal and well verified to ensure the integrity of the global system regardless of any malicious actions taken by the attacker.

Even if a relevant specification were not preinstalled with the vehicle, new safety specifications could cause the vehicle to be updated with the specification at a later date and protect all newly sold vehicles from exhibiting the same problem. With no impact on functionality assuming correct operation of the specification, the costs to test, implement, and distribute the safety updates would be significantly less than that of a dealership-based reflash of the entire ECU, a change directly affecting both the safety and the functionality of the component.

Beyond ensuring the enforcement of functional properties despite a security breach, RV-ECU can also protect the system from a malfunction, helping to curtail automotive recalls. Figure 6 shows an analysis of past automotive recalls. We look only at recalls occurring in the last five years and affecting more than 50k cars, with software errors as the principal contributing factor to the recall.

The results of our initial analysis seem quite promising: simple, English-language properties that are portable across vehicles and manufacturers are often enough to have entirely prevented the recall assuming the presence of a functional runtime verification platform. Of all the recalls we analyzed, only two were not preventable by runtime verification: in both cases, the error causing the recall was a mistake in the specification of the original system rather than in its implementation, meaning that the RV-ECU would potentially enforce incorrect behavior and would not improve or alter the safety of the overall system.

Unlike other implemented and practical systems, our formalisms are quite concise. Figure 7 shows one property featured in Figure 6, namely that the cruise control motor cannot send messages unless the cruise control is operated (started since last stopped). As you can see from the property, a simple regular expression over the cruise control messages and associated recovery action is sufficient to enforce the relevant property in our simulation. Our CAN API provides a standard for translating CAN messages into events, over which the regular expression above is

Automaker	Type	Year	# Vehicles	Relevant Property	Monitoring
Toyota [53]	Mechanical	2010	7.5M	Acceleration messages cannot overlap braking messages on the bus	CAN Only
Chrysler [54]	Software	2015	1.4M	Only messages in scope of the target ECU should be processed	CAN+Local*
Toyota [55]	Software	2015	625K	While moving, the hybrid system can only be shut down through the ignition switch**	CAN Only
Ford [56]	Software	2014	595K	The airbags must deploy within 10 milliseconds of acceleration over a threshold on any axis	Local Only
Ford [57]	Software	2015	432K	The engine cannot be running without the key in the ignition	CAN Only
Honda [58]	Software	2015	92K	Not preventable via RV (specification error)	Local Only
GM [59]	Software	2014	52K	Not preventable via RV (specification error)	Local Only
Jaguar [60]	Software	2011	18K	Cruise control motor cannot send control messages unless cruise control has been started since last stopped	CAN Only

Figure 6: Selected large software recalls since 2010 along with their preventability from monitoring. * = Authentication layer also required, ** = Physical component involved, property may not be sufficient

```
ere : (cruise_control_start cruise_control_message* cruise_control_stop)*

@ fail {
  CAN_DO(CruiseControl, Stop, 1);
}
```

Figure 7: The ERE-based safety property enforcing cruise control messages only while in scope

written. We also provide a function to send messages on the CAN bus, with the component and payload defined as enumerations. Properties can be tested on a PC-based simulation or on-device, as long as specifications for the manufacturer-specific components of the CAN bus are provided. We have reverse engineered several such components on a 2012 Honda Odyssey.

A PRACTICAL DEMONSTRATION

The first step towards demonstrating the separation of functionality and safety on a vehicle architecture using RV-ECU is the creation of a real-vehicle demo showcasing our architecture monitoring a realistic but simplified safety property.

Consider the following body-related property of door safety in a minivan, with electronic controllers that open the rear sliding doors in response to messages over the CAN bus: Unless the driver has unlocked the rear doors from the global vehicle lock/unlock controls, and the doors have not been locked since, the motor responsible for opening the doors should not do so. The safety monitoring code of this property as well as its automatic generation using the technology underlying RV-ECU have been illustrated in Figure 1.

It is not difficult to imagine a situation in which this property could be violated. For example, with a malicious attacker gaining control of only the infotainment system, connected to the body CAN bus, the malicious attacker could easily spoof a “rear door open” message while the vehicle is moving at high speeds to endanger the safety of any potential rear passengers. Alternatively, even in situations where no malicious attacker is present, a malfunctioning ECU connected to the body bus anywhere in the car could create such an unsafe situation by sending a message to the motor to engage. Finally and most likely, a passenger seating in the rear seat may (mistakenly) push the door open button, which subsequently sends the motor engage message. The last scenario above is obviously checked by almost all cars, likely using a protocol implemented in the door ECU that sends data-collecting messages to other ECUs and then sends the motor engage message only if it determines it is safe to do so. Not only is the door ECU more complex than needs to be due to mixing functionality and safety, but the overall systems is still unsafe, because the other two scenarios can still happen. With RV-ECU, all three scenarios above are treated the same way, with a global monitor ECU in charge of monitoring the safety property possibly among tens or hundreds of other similar properties, and with any other ECU free of developer-provided safety checking code.

Figure 8: RV-ECU development prototype connected to the body CAN bus of a 2012 Honda Odyssey

The overall system is simpler and safer.

We have obtained a STM3210C-EVAL development board implementing the popular STM32 embedded architecture. We are mimicking a minimal AUTOSAR-like API exclusively for interacting with the CAN bus, and running our certifiable high-assurance code to monitor and enforce the previously mentioned property in a 2012 Honda Odyssey minivan. Our demo is implemented and working, and we intend to demonstrate it as part of our presentation in SAE 2016. Figure 8 shows our development embedded board running on the CAN bus of our demo vehicle, attached through a connection in the driver’s side lock control unit. Figure 9 shows the FSM-based property we monitor in our initial demonstration of the body CAN, available at https://runtimeverification.com/ecu. In English, the property states that the headlights should be on whenever the windshield wipers are on, and set to the user’s selected mode when the wipers are off. While this is likely not an entirely realistic property (as manufacturers wish to grant users the ultimate control over headlight state), it serves as a good demonstration for the ability of our monitoring platform to enforce real properties on the global CAN bus.

Despite the simplicity of this formalism, the need to maintain regularity imposed by using an FSM render the property quite verbose. As you can see, for a simple one-line English property, the property monitored in a vehicle is over 20 lines in length. Still, the property is relatively simple to understand and create: we have one state for each possible (wiper, headlight) state pair, where the wipers and headlights can either be on or off. In this example, when we refer to headlights we are referring to the standard night time low beams of our test vehicle. We then have one transition from each state for each possible change in state of the subcomponents. For example, if the wipersOn

```
fsm :
 wipersOffHeadlightsOff [
   wipersOn -> wipersOnHeadlightsOff,
   wipersOff -> wipersOffHeadlightsOff,
   headlightsOn -> wipersOffHeadlightsOn,
   headlightsOff-> wipersOffHeadlightsOff
 ]
 wipersOffHeadlightsOn [
   wipersOn -> wipersOnHeadlightsOn,
   wipersOff -> wipersOffHeadlightsOn,
   headlightsOn -> wipersOffHeadlightsOn,
   headlightsOff-> wipersOffHeadlightsOff
 ]
 wipersOnHeadlightsOn [
   wipersOn -> wipersOnHeadlightsOn,
   wipersOff -> wipersOffHeadlightsOn,
   headlightsOn -> wipersOnHeadlightsOn,
   headlightsOff -> wipersOnHeadlightsOff
 ]
 wipersOnHeadlightsOff [
   wipersOn -> wipersOnHeadlightsOff,
   wipersOff -> wipersOffHeadlightsOff,
   headlightsOn -> wipersOnHeadlightsOn,
   headlightsOff -> wipersOnHeadlightsOff
 ]

 @wipersOnHeadlightsOff {
   CAN_DO(Headlight, High, 1);
 }
```

Figure 9: The FSM-based safety property experimentally enforced on the 2012 Odyssey

event is seen in the wipersOffHeadlightsOff state, we transition to the wipersOnHeadlightsOff state.

Lastly, we have an unsafe state (wipersOnHeadlightsOff), and a recovery handler (@wipersOnHeadlightsOff) which sends a message to the CAN bus using our built-in CAN communication API to turn the Headlight component to High one time (`CAN_DO(Headlight, High, 1);`) any time the associated state is entered.

This recovery means that the unsafe state will never be the permanent state of the system. The transition out of the unsafe state is driven by a response from the monitor to the transition into the state, showing the possibility of recovering from property violations using only bus messages. On a real vehicle, the effect of running this monitoring code on our prototype RV-ECU which is connected to the CAN bus is that it is impossible to turn the wipers on without having the headlights turn on, regardless of the position of the headlight controls. User control of the headlights is also returned any time the wipers turn off, and the monitor enters a set of states it knows to be safe (as any states with wipers off are known to be safe).

This property can alternatively be specified in the more concise past time linear temporal logic (PTLTL) formalism, also supported by RV-Monitor. In this formalism, the formal definition of the property would be: [](wipersOn => (headlightsOff, headlightsOn]). This property states that it is always the case that when the wipers are on, the headlights have not been turned off since they have been turned on (using interval notation). While we do support this notation in RV-Monitor and this representation showcases our ability to concisely express formal specifications, we believe that past-time linear temporal logic is beyond the immediate familiarity level with formal properties of the majority of our target with the RV-ECU product. We will thus focus primarily on the familiar FSM and regular expression formalisms, common in general practice.

FUTURE WORK AND APPLICATIONS

One unanswered research question regarding the proposed RV-ECU safety architecture, shared with other formal analysis methods in the automotive domain, is what is the ideal formalism suited for mathematically defining automotive properties. Runtime Verification, Inc., will work with their automotive partners and customers to provide an intuitive domain-specific formal representation and associated plugin for our system allowing safety engineers or managers to comfortably specify such properties, lowering the barrier to entry for our technology and facilitating its uptake in industry. Such a plugin would likely also support the definition of real-time and temporal safety properties to fully specify the range of possible safety specifications associated with a safety-critical real time system.

As part of this process, we are seeking an automotive manufacturer or supplier willing to experiment with our technology in their development environment, evaluating the benefits of our specification language, code generation infrastructure, and the general separation of safety and functionality we provide to the specification and monitoring of complex software systems.

TECHNICAL LIMITATIONS AND DRAWBACKS

There are several limitations and drawbacks raised by the potential vehicle architecture we propose. The

first is the additional communications on the CAN bus required between the local and global monitors. In the proposed architecture, traffic can be as much as doubled for safety critical messages when the RV-ECU acts as a relay point, as compared to when safety is not checked at all. In an already overcrowded bus, such limitations could be prohibitive to the implementation of our solution. To mitigate this, it is possible to monitor properties primarily locally, monitoring only properties involving multiple ECUs through the global safety monitor. It is worth noting that existing architectures also require a number of messages to be sent specifically for checking safety, e.g., the door ECU requesting data from other ECUs when the door open button is pushed; more research is needed to compare the number of messages that RV-ECU requires versus the existing architectures. In the long term, our hope is that efforts aiming to replace the CAN bus with faster and higher-throughput communication standards will allow for our additional communication without overburdening the system.

Another key technical challenge for our technology is developing a formalism and infrastructure capable of handling the real-time properties required by the automotive industry. Because we have no access to the proprietary specifications currently used, we are unable to develop such a system. We thus wish to start with an architecture capable of handling non-real-time properties, extending it with real time support as is required to handle the needs of our customers.

The main risk in the adoption and development of runtime verification in automotive however lies in the development of accurate, rigorous specifications which the automotive industry does not currently have in the development process. With only a vague, often informal notion of formal system safety, the majority of OEMs and suppliers have not fully and rigorously defined precisely what the safety of a vehicle system consists of. This initial effort to formalize the notion of safety in the vehicle may be cost prohibitive and difficult, but remains necessary for the eventual creation of a system with strong safety guarantees and high assurance. We believe this undertaking will have a positive effect on the automotive industry, providing a rigorous notion and understanding of what safety means in the context of the vehicle system. This rigorous notion will help at every level of the development cycle, facilitating testing, development of new functionality, and regulatory certification.

One further and clear technical limitation of our approach is its inability to protect from hardware faults. Because our approach operates at the software level, any flaws in the CAN driver being used or the hardware of any individual ECU can still cause problems undetectable and unforeseen by the specification monitoring system. While the former can be mitigated by full verification of the CAN driver, a more traditional fault detection approach is likely more suitable for detecting faults in the actuators, sensors, and processing hardware involved in the vehicle system.

It is also important to note that extensive full-vehicle testing will still be required despite the presence of our safety architecture. The effects of our monitoring code and the effects of the interactions of the specification monitors with the full system cannot be determined without testing. We hope that with rigorous, checkable specifications and descriptive error conditions, our system will speed the testing cycle for safety requirements by allowing rapid evaluation of the system against its stated requirements. Despite this, rigorous conventional testing is still required to maintain the safety of the full vehicle system.

Lastly, there is a risk that our specifications will themselves introduce safety risks in the system: if the specifications are inaccurate, unforeseen circumstances can create unexpected programmatic behaviors actually detrimental to the safety of the system. For example, in Figure 4, a monitor could theoretically override or exclude a message by the controller it incorrectly believes to be unsafe, which would itself cause safety problems in the vehicle. While this is undoubtedly possible, our belief is that any unforeseen behaviors in the formal specifications provided could just as readily be present in the code itself, which implements informal specifications. Formalizing the specifications implicit in the current codebase rigorously will not inherently introduce unforeseen behaviors, and we expect that such formal rigor in the testing phase will actually help reveal previously unconsidered safety-critical interactions. In the cases where there are unintended interactions between the monitors and the system itself, traditional testing should be able to reveal them at least as readily as it reveals inconsistencies between actual and expected behavior in current systems.

CONCLUSION

Thus, we claim that separating safety and functionality in the modern automobile system would help find software bugs early in development, avoid recalls, and improve communication between original equipment manufacturers and their suppliers. We propose runtime verification as one solution allowing for this separation, and introduce a potential architecture for realizing such a practical separation.

We see that specifications checked at runtime can be

both concise and formally precise, allowing for their development by engineers and managers not trained in formal methods while ensuring they are modular and easily sharable. We implement such a system with a practical demonstration of a simplified body safety property, and lay out the roadmap for future work enabled by the separation of safety and functionality. We discuss the technical limitations and drawbacks of our approach, including resource overhead, incomplete specifications, and an inability to deal with low level hardware faults.

Overall, we seek to develop a commercial product usable by the automotive industry to add runtime verification to vehicles, both in the testing and development cycles and in production. We have already created a production-ready architecture for the provably correct monitoring of safety properties on automotive buses, and intend to partner with interested parties towards the application of such a system. We would like to apply such a technology to large-scale projects to analyze scaling concerns and demonstrate the feasibility of our approach in production, improving the overall safety of automotive systems.

ACKNOWLEDGEMENTS

We would like to thank our partners at Toyota Info Technology Center, Inc., and DENSO International America, Inc., for their collaboration, industry insight, and generous funding. We would also like to thank the National Science Foundation for funding work related to this project under SBIR (small business development) grants, specifically related to the analysis of past recalls, development of a prototype, and formal verification of runtime monitors. Additionally, we would like to thank NASA for SBIR funding that has helped improve the infrastructure of our code prover.

Further assistance for this work was provided by research and development performed at the Formal Systems Laboratory of the University of Illinois at Urbana-Champaign, including funding from NSF, NASA, DARPA, NSA, and Boeing grants.

References

[1] Robert N Charette. “This car runs on code.” In: *IEEE Spectrum* 46.3 (2009), p. 3.

[2] Shinichi Shiraishi and Mutsumi Abe. “Automotive system development based on collaborative modeling using multiple adls.” In: *ESEC/FSE 2011 (Industial Track)* (2011).

[3] Larry P. Vellequette. *Fiat Chrysler recalls 1.4 million vehicles to install anti-hacking software*. http://www.autonews.com/article/20150724/OEM11/150729921/fiat-chrysler-recalls-1.4-million-vehicles-to-install-anti-hacking.

[4] Christiaan Hetzner. *VW ordered to recall 2.4 million cars in Germany with cheat software*. http://www.autonews.com/article/20151015/COPY01/310159986/vw-ordered-to-recall-2-4-million-cars-in-germany-with-cheat-software.

[5] Reuters. *Toyota recalls 625,000 cars over software malfunction*. http://www.dw.com/en/toyota-recalls-625000-cars-over-software-malfunction/a-18585121.

[6] Associated Press. *Ford Recalls 432,000 Cars Over Software Problem*. http://www.dailyfinance.com/2015/07/02/ford-recalls-cars-software-problem/.

[7] Associated Press. *Honda recalling 143,000 Civics, Fits to fix faulty software*. http://bigstory.ap.org/article/2f5f75fd91e64ec6bde06bacf9824867/honda-recalling-143000-civics-fits-fix-faulty-software.

[8] Eric Beech. *GM recalls nearly 52,000 SUVs for inaccurate fuel gauge*. http://www.reuters.com/article/2014/05/03/us-gm-recall-suv-idUSBREA4209C20140503.

[9] Ben Klayman. *Chrysler recalls 18,092 Fiat 500L cars for transmission issue*. http://www.reuters.com/article/2014/03/17/us-chrysler-usrecall-idUSBREA2G0PU20140317.

[10] Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. “Test Suites for Benchmarks of Static Analysis Tools.” In: *ISSRE 15 Industry Track*. NIST. 2015.

[11] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. Vol. 26202649. MIT press Cambridge, 2008.

[12] Dawson Engler and Madanlal Musuvathi. “Static Analysis Versus Software Model Checking for Bug Finding.” In: *In VMCAI*. Springer, 2004, pp. 191–210.

[13] *Java Path Finder*. http://babelfish.arc.nasa.gov/trac/jpf. Accessed: 2014-05-17.

[14] *NuSMV Home Page*. http://nusmv.fbk.eu/. Accessed: 2014-05-17.

[15] *UPPAAL: Academic Home*. http://www.uppaal.org/. Accessed: 2014-05-17.

[16] Klaus Havelund and Grigore Rosu. “Preface: Volume 55, Issue 2.” In: *Electronic Notes in Theoretical Computer Science* 55.2 (2001), pp. 287–288.

[17] E. Allen Emerson and Edmund M. Clarke. “Characterizing Correctness Properties of Parallel Programs Using Fixpoints.” In: *Proceedings of the 7th Colloquium on Automata, Languages and Programming*. Springer-Verlag, 1980, pp. 169–181. ISBN: 3-540-10003-2.

[18] Klaus Havelund and Grigore Rosu. “Monitoring Programs Using Rewriting.” In: *16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA*. IEEE Computer Society, 2001, pp. 135–143. ISBN: 0-7695-1426-X. DOI: 10.1109/ASE.2001.989799.

[19] Amir Pnueli and Aleksandr Zaks. “PSL Model Checking and Run-Time Verification Via Testers.” In: *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*. Ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. Vol. 4085. Lecture Notes in Computer Science. Springer, 2006, pp. 573–586. ISBN: 3-540-37215-6. DOI: 10.1007/11813040_38.

[20] Andreas Bauer, Martin Leucker, and Christian Schallhart. “The Good, the Bad, and the Ugly, But How Ugly Is Ugly?” In: *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*. Ed. by Oleg Sokolsky and Serdar Tasiran. Vol. 4839. Lecture Notes in Computer Science. Springer, 2007, pp. 126–138. ISBN: 978-3-540-77394-8. DOI: 10.1007/978-3-540-77395-5_11.

[21] Howard Barringer, Klaus Havelund, David E. Rydeheard, and Alex Groce. “Rule Systems for Runtime Verification: A Short Tutorial.” In: *Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers*. 2009, pp. 1–24. DOI: 10.1007/978-3-642-04694-0_1.

[22] Feng Chen and Grigore Rosu. “Parametric Trace Slicing and Monitoring.” In: *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by Stefan Kowalewski and Anna Philippou. Vol. 5505. Lecture Notes in Computer Science. Springer, 2009, pp. 246–261. ISBN: 978-3-642-00767-5. DOI: 10.1007/978-3-642-00768-2_23.

[23] Howard Barringer and Klaus Havelund. “Internal versus External DSLs for Trace Analysis - (Extended Abstract).” In: *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*. 2011, pp. 1–3. DOI: 10.1007/978-3-642-29860-8_1.

[24] Moonjoo Kim, Insup Lee, Usa Sammapun, Jangwoo Shin, and Oleg Sokolsky. “Monitoring, Checking, and Steering of Real-Time Systems.” In: *Electr. Notes Theor. Comput. Sci.* 70.4 (2002), pp. 95–111. DOI: 10.1016/S1571-0661(04)80579-6.

[25] Eric Bodden, Laurie J. Hendren, Patrick Lam, Ondrej Lhoták, and Nomair A. Naeem. “Collaborative Runtime Verification with Tracematches.” In: *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*. Ed. by Oleg Sokolsky and Serdar Tasiran. Vol. 4839. Lecture Notes in Computer Science. Springer, 2007, pp. 22–37. ISBN: 978-3-540-77394-8. DOI: 10.1007/978-3-540-77395-5_3.

[26] Feng Chen and Grigore Rosu. “Mop: an efficient and generic runtime verification framework.” In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. Ed. by Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. ACM, 2007, pp. 569–588. ISBN: 978-1-59593-786-5. DOI: 10.1145/1297027.1297069.

[27] Volker Stolz and Eric Bodden. “Temporal Assertions using AspectJ.” In: *Electr. Notes Theor. Comput. Sci.* 144.4 (2006), pp. 109–124. DOI: 10.1016/j.entcs.2006.02.007.

[28] Justin Seyster, Ketan Dixit, Xiaowan Huang, Radu Grosu, Klaus Havelund, Scott A. Smolka, Scott D. Stoller, and Erez Zadok. “Aspect-Oriented Instrumentation with GCC.” In: *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*. 2010, pp. 405–420. DOI: 10.1007/978-3-642-16612-9_31.

[29] Mads Dam, Bart Jacobs, Andreas Lundblad, and Frank Piessens. “Security Monitor Inlining for Multithreaded Java.” In: *Genoa: Proceedings of the 23^{rd} European Conference on ECOOP — Object-Oriented Programming*. 2009, pp. 546–569. ISBN: 978-3-642-03012-3.

[30] Irem Aktug, Mads Dam, and Dilian Gurov. “Provably Correct Runtime Monitoring.” In: *FM ’08: Proceedings of the 15^{th} int. symposium on Formal Methods*. Turku, Finland, 2008, pp. 262–277. ISBN: 978-3-540-68235-6.

[31] Úlfar Erlingsson and Fred B. Schneider. “SASI enforcement of security policies: a retrospective.” In: *NSPW ’99: workhop on New security paradigms*. 2000, pp. 87–95. ISBN: 1-58113-149-6.

[32] Horatiu Cirstea, Pierre-Etienne Moreau, and Anderson Santana de Oliveira. “Rewrite Based Specification of Access Control Policies.” In: *Electron. Notes Theor. Comput. Sci.* 234 (2009), pp. 37–54. ISSN: 1571-0661. DOI: http://dx.doi.org/10.1016/j.entcs.2009.02.071.

[33] Anderson Santana de Oliveira, Eric Ke Wang, Claude Kirchner, and Hélène Kirchner. “Weaving rewrite-based access control policies.” In: *FMSE’07: Proceedings of the ACM workshop on Formal Methods in Security Engineering*. 2007, pp. 71–80.

[34] Yliès Falcone, Sebastian Currea, and Mohamad Jaber. “Runtime verification and enforcement for Android applications with RV-Droid.” In: *Runtime Verification*. Springer. 2013, pp. 88–95.

[35] *ProofObjects: Working with Explicit Evidence in Coq*. http://www.cs.cornell.edu/~clarkson/courses/sjtu/2014su/terse/ProofObjects.html. Accessed: 2015-09-1.

[36] *How I discovered CCS Injection Vulnerability (CVE-2014-0224)*. http://ccsinjection.lepidum.co.jp/blog/2014-06-05/CCS-Injection-en/index.html. Accessed: 2014-05-15.

[37] *Formalizing 100 theorems in Coq*. http://perso.ens-lyon.fr/jeanmarie.madiot/coq100/. Accessed: 2015-09-1.

[38] *CompCert - The CompCert C Compiler*. http://compcert.inria.fr/compcert-C.html. Accessed: 2015-09-1.

[39] Xavier Leroy. *The CompCert C Verified Compiler*. 2012.

[40] Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. “Formal C semantics: CompCert and the C standard.” In: *Interactive Theorem Proving*. Springer, 2014, pp. 543–548.

[41] Rui Zhou, Rong Min, Qi Yu, Chanjuan Li, Yong Sheng, Qingguo Zhou, Xuan Wang, and Kuan-Ching Li. “Formal verification of fault-tolerant and recovery mechanisms for safe node sequence protocol.” In: *Advanced Information Networking and Applications (AINA), 2014 IEEE 28th International Conference on*. IEEE. 2014, pp. 813–820.

[42] Thomas Kropf. *Introduction to formal hardware verification*. Springer Science & Business Media, 2013.

[43] Diogo Behrens, Stefan Weigert, and Christof Fetzer. “Automatically tolerating arbitrary faults in non-malicious settings.” In: *Dependable Computing (LADC), 2013 Sixth Latin-American Symposium on*. IEEE. 2013, pp. 114–123.

[44] Aaron Kane. “Runtime Monitoring for Safety-Critical Embedded Systems.” In: (2015).

[45] Lee Pike, Alwyn Goodloe, Robin Morisset, and Sebastian Niller. “Copilot: a hard real-time runtime monitor.” In: *Runtime Verification*. Springer. 2010, pp. 345–359.

[46] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. “Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor.” In: *USENIX Security Symposium*. San Diego, USA. 2004, pp. 179–194.

[47] Rodolfo Pellizzoni, Patrick Meredith, Marco Caccamo, and Grigore Rosu. *Bus-MOP: a runtime monitoring framework for PCI peripherals*. Tech. rep. Technical report, University of Illinois at Urbana-Champaign, 2008. Available at http://netfiles. uiuc. edu/rpelliz2/www.

[48] Rodolfo Pellizzoni, Patrick Meredith, Marco Caccamo, and Grigore Rosu. “Hardware runtime monitoring for dependable cots-based real-time embedded systems.” In: *Real-Time Systems Symposium, 2008*. IEEE. 2008, pp. 481–491.

[49] Andy Greenberg. *Hackers Remotely Kill a Jeep on the HighwayWith Me in It*. http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/.

[50] Charlie Miller and Chris Valasek. *Remote Exploitation of an Unaltered Passenger Vehicle*. www.illmatics.com/Remote%20Car%20Hacking.pdf. Accessed: 2015-08-14.

[51] *Remote Exploitation of an Unaltered Passenger Vehicle*. https://www.youtube.com/watch?v=0obLb1McxnI.

[52] Douglas Salane. “Are Large Scale Data Breaches Inevitable?” In: *Cyber Infrastructure Protection* 9 (2009).

[53] National Highway Traffic Safety Administration. *Consumer Advisory: Toyota Owners Advised of Actions to Take Regarding Two Separate Recalls*. 2010. URL: \url{http://www.nhtsa.gov/CA/02-02-2010}.

[54] National Highway Traffic Safety Administration. *Part 573 Safety Recall Report 15V-461*. 2015. URL: \url{https://www-odi.nhtsa.dot.gov/acms/cs/jaxrs/download/doc/UCM483036/RCLRPT-15V461-9407.pdf}.

[55] National Highway Traffic Safety Administration. *RECALL Subject : Inverter Failure may cause Hybrid Vehicle to Stall — NHTSA*. 2015. URL: \url{http://www-odi.nhtsa.dot.gov/owners/SearchResults?searchType=ID&targetCategory=R&searchCriteria.nhtsa_ids=15V449000}.

[56] National Highway Traffic Safety Administration. *RECALL Subject : Side-Curtain Rollover Air Bag Deployment Delay — NHTSA*. 2014. URL: \url{http://www-odi.nhtsa.dot.gov/owners/SearchResults?refurl=email&searchType=ID&targetCategory=R&searchCriteria.nhtsa_ids=14V237}.

[57] Chicago Sun Times. *Ford recalls 432,000 cars because of software problem*. 2015. URL: \url{http://chicago.suntimes.com/business/7/71/740316/ford-recalls-software-problem}.

[58] National Highway Traffic Safety Administration. *RECALL Subject : Incorrect Yaw Rate/FMVSS 126 — NHTSA*. 2015. URL: \url{http://www-odi.nhtsa.dot.gov/owners/SearchResults.action?searchType=ID&targetCategory=R&searchCriteria.nhtsa_ids=13V157}.

[59] National Highway Traffic Safety Administration. *RECALL Subject : Inaccurate Fuel Gauge Reading — NHTSA*. 2014. URL: \url{http://www-odi.nhtsa.dot.gov/owners/SearchResults?searchType=ID&targetCategory=R&searchCriteria.nhtsa_ids=14V223000}.

[60] BBC News. *Jaguar recalls 18,000 cars over 'faulty' cruise control*. 2011. URL: \url{http://www.bbc.com/news/business-15410253}.

[61] Oleg Sokolsky and Serdar Tasiran, eds. *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*. Vol. 4839. Lecture Notes in Computer Science. Springer, 2007. ISBN: 978-3-540-77394-8.